



**POLYTECHNIQUE
MONTRÉAL**

UNIVERSITÉ
D'INGÉNIERIE

PHS3910 – TECHNIQUES EXPÉRIMENTALES ET INSTRUMENTATION

Équipe : Lundi 03

Écran tactile acoustique

Fiche technique du prototype

Présenté à
Jean Provost
Lucien Weiss

Par :
Émile **Guertin-Picard** (2208363)
Philippine **Beaubois** (2211153)
Marie-Lou **Dessureault** (2211129)
Maxime **Rouillon** (2213291)

20 octobre 2024
Département de Génie Physique
Polytechnique Montréal

Table des matières

1	Description générale et spécifications	1
2	Rapports de tests	2
2.1	Nombre de bits des échantillons du signal	2
2.2	Fréquence d'échantillonnage du signal	2
2.3	Contenu fréquentiel du signal	3
2.4	Interdépendance des facteurs	3
2.5	Incertitudes	6
3	Codes	7
3.1	Programme pour modifier le contenu fréquentiel et la fréquence d'échantillonnage	7
3.2	Programme pour changer le nombre de bits des échantillons	10
3.3	Programme pour calculer la résolution et le contraste pour un dictionnaire de notes	11
3.4	Programme pour enregistrer un dictionnaire de notes	15
3.5	Programme pour jouer du piano en temps réel	16

1 Description générale et spécifications

Cette fiche technique présente les caractéristiques d'un piano construit avec un écran tactile acoustique. Un capteur piezoélectrique, sur une plaque de plexiglas de 5 mm d'épaisseur, localise un impact par son onde sonore, pour permettre de jouer la note appropriée en temps réel. Cette plaque et ses dimensions sont présentées à la figure 1. Le piano peut jouer une seule gamme (12 notes), et est limité à ne pouvoir jouer qu'une seule note à la fois. L'acquisition de signal sonore se fait à une fréquence de 44100 Hz par un ADC, donnant des échantillons de 32 bits. Des fréquences sonores de 0 Hz à 22000 Hz sont présentes. Le délai entre la frappe et le son de la note est d'environ 200 ms, dépendant de la puissance de l'ordinateur qui lit les données du capteur.

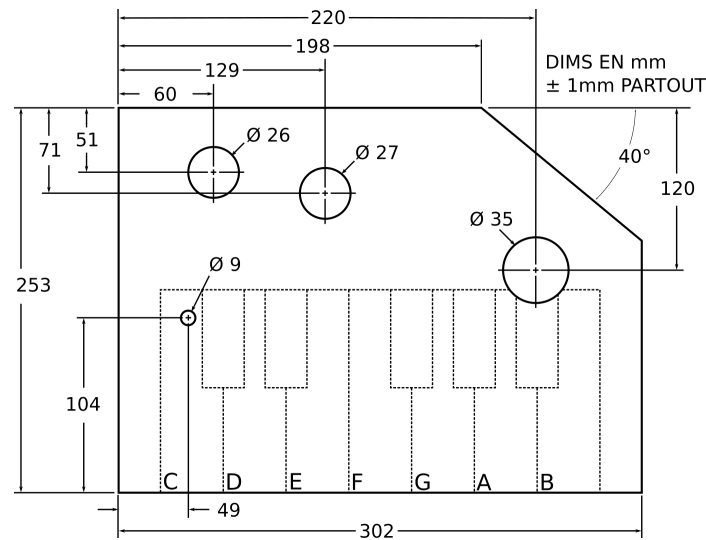


Figure 1 : Schéma avec dimensions du prototype de piano tactile. Le capteur piezoélectrique a son centre de positionné à 176 mm du côté gauche et à 207 mm du bas de la plaque approximativement.

Le tableau 1 résume les résultats pertinents des tests de caractérisations effectués sur le prototype. La première ligne du tableau illustre la résolution, grandeur qu'une note doit prendre pour être distincte, et le contraste, qui quantifie la différence entre les signaux obtenus, pour le piano présenté ci-dessus. De là, des bornes ont été posées sur le nombre de bits, la fréquence d'échantillonnage et le contenu fréquentiel pour évaluer l'influence de chacun sur la performance globale du dispositif. Ces tests ont été effectués dans l'optique d'amoinrir les coûts pour un deuxième prototype. Les tests choisis dans le tableau 1 sont ceux pour lesquels la dégradation des facteurs résultaient en valeurs convenables de résolution et de contraste.

Bits	Facteurs		Résolution (cm)	Contraste
	Fréquence d'échantillonnage (Hz)	Contenu fréquentiel (Hz)		
32	44100	-	6.39 ± 0.08	0.62 ± 0.02
16	44100	-	5.0 ± 0.2	0.70 ± 0.08
8	44100	-	4.2 ± 0.2	0.58 ± 0.09
32	44100	300-5000	6.71 ± 0.08	0.529 ± 0.004
32	44100	300-1500	6.27 ± 0.08	0.72 ± 0.01
32	1764	300-1500	4.2 ± 0.1	0.577 ± 0.003
32	1764	100-1500	5.05 ± 0.08	0.57 ± 0.02

Table 1 – Tableau des spécifications de l'écran tactile acoustique avec modification de facteurs.

2 Rapports de tests

Pour pouvoir réduire les coûts blabla... **TODO**

2.1 Nombre de bits des échantillons du signal

Pour modifier le nombre de bits artificiellement de chaque signal, le processus est assez simple. Le nombre de bits disponibles doit être déterminé au début pour la précision. En effet, lorsque le nombre de bits est réduit, il n'est pas possible de tous les utiliser pour la précision. Dans les faits, un bit est toujours réservé pour la description du signe (positif ou négatif). Puisque la base 2 est utilisée dans ce prototype, le nombre de niveaux disponibles pour approximer les données est le suivant : 2^{nbr_bit-1} . Ainsi, pour diminuer la précision des valeurs du signal au nombre de bits souhaité, le signal sera multiplié par le facteur suivant,

$$Facteur = \frac{2^{nbr_bit-1}}{2} \quad (1)$$

pour normaliser le signal, ensuite chaque valeur sera arrondie à son niveau le plus proche. Le signal sera alors divisé par *Facteur* pour revenir à son échelle d'origine. Avec cette méthode, les résultats suivants ont été obtenus et sont présentés dans le tableau 2.

Dictionnaire	Résolution (cm)	Contraste
1bit	0 ± 8	0 ± 100
2bit	0.0 ± 0.2	0 ± 5
3bit	0.04 ± 0.06	0 ± 2
4bit	0.04 ± 0.03	0 ± 1
6bit	0.042 ± 0.006	0.6 ± 0.3
8bit	0.042 ± 0.002	0.58 ± 0.09
16bit	0.050 ± 0.002	0.70 ± 0.08
Original	0.064 ± 0.001	0.62 ± 0.02

Table 2 – Tableau des résultats

En analysant les données présentes dans ce tableau, on constate qu'en dessous de 6 bits, les valeurs de résolution et de contraste commencent à être nettement moins correctes. Cette affirmation peut être démontrée par la nette augmentation des incertitudes, ce qui rend les valeurs plus incertaines. Il est donc juste de déterminer qu'un signal traité avec moins de ressources que celles requises pour 6 bits aurait un impact néfaste sur l'efficacité du fonctionnement du piano. Cependant, cela nous permet également de remarquer qu'un prototype pourrait fonctionner avec moins de bits que le prototype actuel. Cela pourrait être bénéfique car il permettrait de faire fonctionner correctement un prototype utilisant moins de mémoire et donc moins de ressources. Cela pourrait permettre de réaliser un projet similaire mais a moindre couts.

2.2 Fréquence d'échantillonnage du signal

Pour tester différentes fréquences d'échantillonnage plus faibles que la fréquence initiale, il suffit d'échantillonner l'échantillon déjà obtenu. En effet, en prenant un point sur deux dans un échantillon dont la fréquence d'échantillonnage est de 44 100 Hz, le nouvel échantillon a directement une fréquence d'échantillonnage de 22 050 Hz.

En procédant de cette façon, plusieurs fréquences d'échantillonnage ont été artificiellement enregistrées. Les résultats de résolution et contraste qu'elles ont produit est présenté au tableau 2.

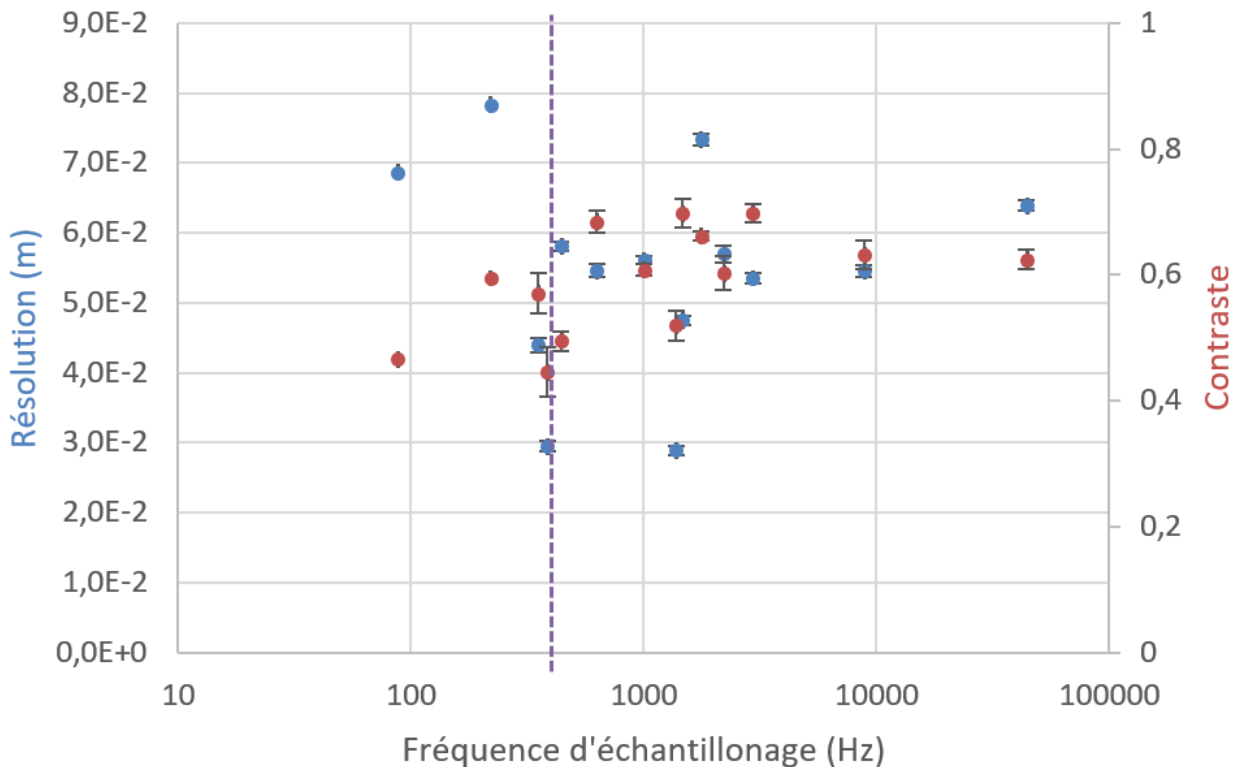


Figure 2 : Contraste et résolution en fonction de la fréquence d'échantillonnage

26 valeurs de fréquences testées. Seules les valeurs non-aberrantes sont présentes sur le graphique. La ligne pointillée verticale indique la moitié des valeurs testées. On constate que peu de valeurs sont affichées pour une fréquence en dessous de 400 Hz. Cela est dû à des aberrations dans la convergence du fit gaussien utilisé dans le code, ce qui témoigne de données d'une qualité insuffisante pour être utilisées dans la conception d'un piano. On remarque également que la fréquence et le contraste varient légèrement entre 400 Hz et 44,1 kHz, mais qu'il n'y a pas de tendance à la hausse ou à la baisse générale. Cela indique qu'on peut diminuer la fréquence d'échantillonnage jusqu'à 400 Hz sans trop d'impact sur la résolution et le contraste.

2.3 Contenu fréquentiel du signal

Pour stocker le signal, il est possible de le transformer dans le domaine fréquentiel, puis d'en retirer une partie avant de l'enregistrer. De cette façon, un moins grand nombre de bit est nécessaire l'information.

Le processus pour retirer une partie du contenu fréquentiel du signal consiste à lui appliquer une FFT, puis à définir des filtres à appliquer, d'obtenir le signal fréquentiel filtré et enfin appliquer la FFT inverse pour retrouver un signal dans le domaine temporel ayant un contenu fréquentiel réduit. Il est alors possible de tester la qualité de ce nouveau signal en calculant la résolution et le contraste obtenu à l'aide d'une corrélation avec un ensemble de vecteurs ayant été réduits de la même façon.

2.4 Interdépendance des facteurs

L'interdépendance des facteurs a dû être prise en compte lors de la caractérisation du prototype. Sachant que le nombre de bits des échantillons affecte principalement l'analyse des signaux, il a été convenu que son

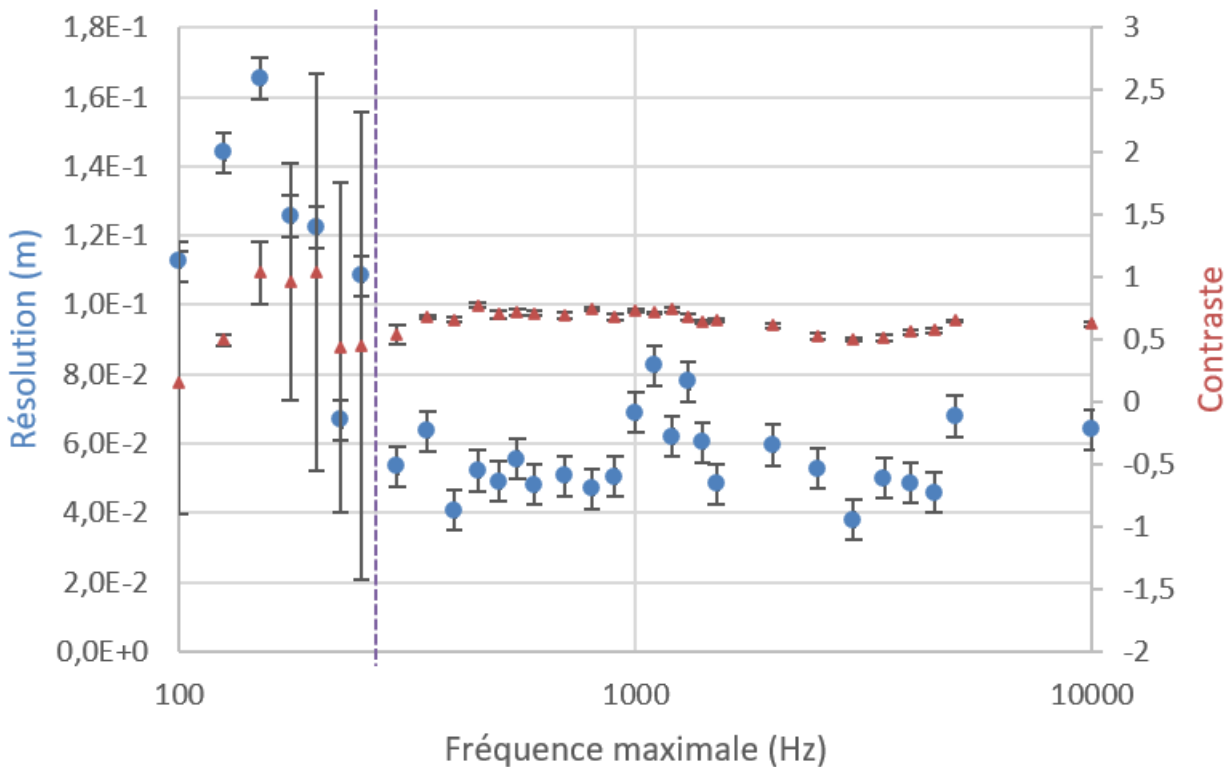


Figure 3 : Contraste et résolution en fonction de la fréquence maximale conservée

influence pouvait être analysée de manière indépendante aux deux autres facteurs. Cependant, la fréquence d'échantillonnage et le contenu fréquentiel étaient soupçonnés d'avoir une interdépendance non négligeable sur les valeurs de contraste et de résolution obtenues. Effectivement, l'impact de la fréquence de Nyquist du signal pouvait potentiellement faire en sorte qu'un signal ayant une fréquence d'échantillonnage plus basse fonctionne tout de même correctement en diminuant la borne supérieure du contenu fréquentiel. La théorie de Nyquist stipule que la fréquence d'échantillonnage minimale d'un signal doit nécessairement correspondre au double de la fréquence maximale du signal évalué [1].

Pour ce faire, certaines plages de contenu fréquentiel ont été choisies, pour ensuite évaluer l'impact de la réduction de la fréquence d'échantillonnage. Des tests ont été effectués pour les bornes supérieures suivantes : 5000 Hz, 3000 Hz, 2250 Hz, 1750 Hz, 1500 Hz et 1000 Hz. La borne inférieure a été maintenue à 300 Hz, ce qui correspond à une limite satisfaisante, tel que déterminé dans les autres tests. Pour essayer de visualiser l'influence de la fréquence de Nyquist, les figures 5 et 6 illustrent les résultats pour la résolution et le contraste.

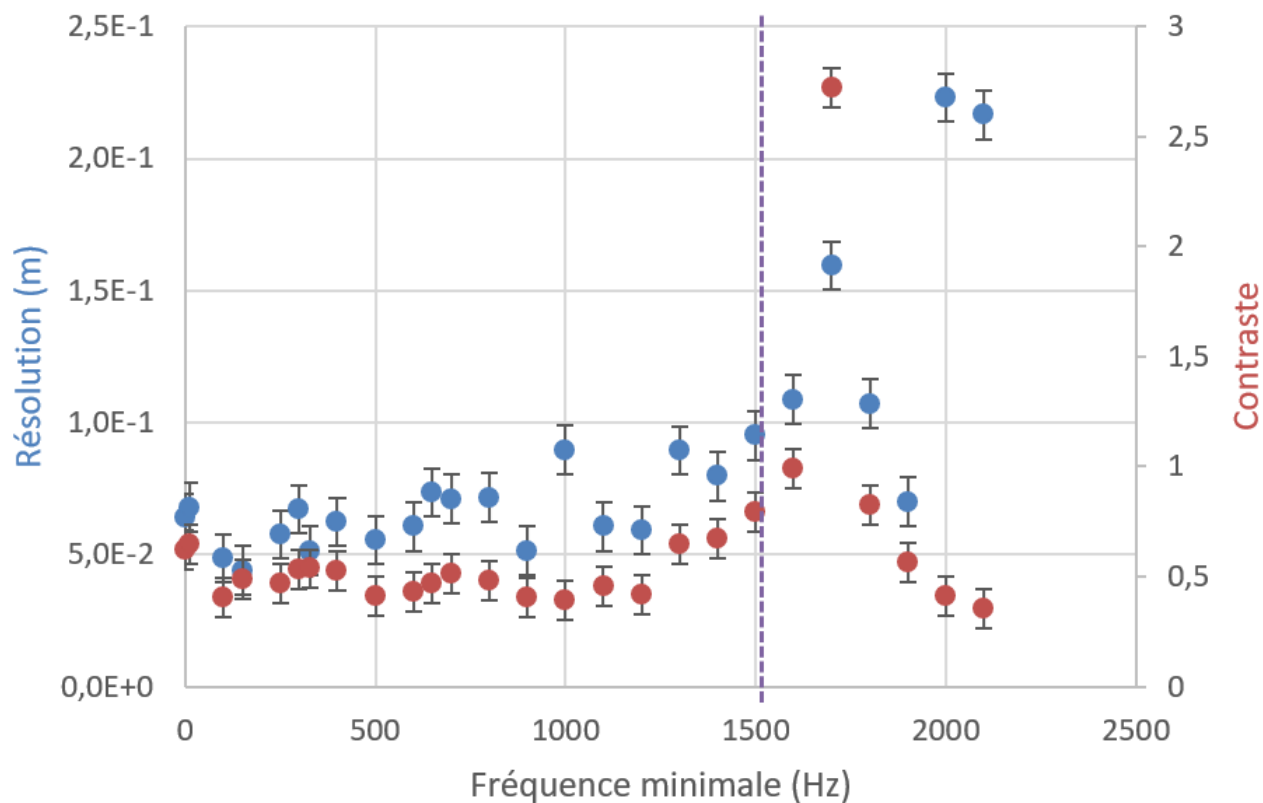


Figure 4 : Contraste et résolution en fonction de la fréquence minimale conservée

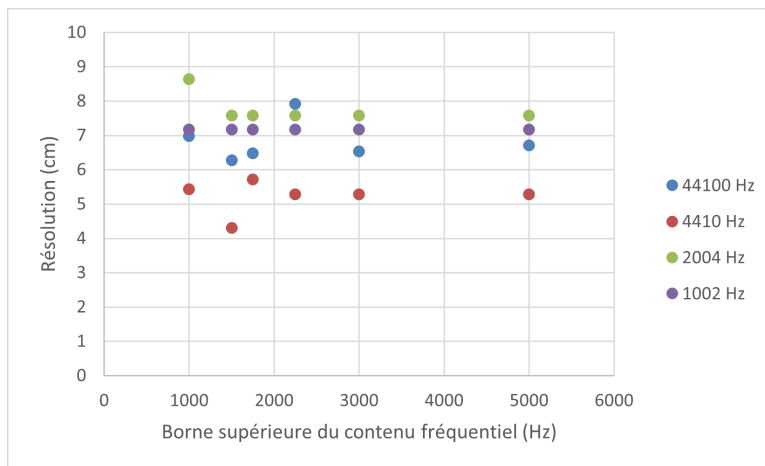


Figure 5 : Résolution en fonction de la fréquence d'échantillonnage (1002, 2004, 4410 et 44100 Hz) et de la fréquence de la borne supérieure du contenu fréquentiel. Ici, la borne inférieure est maintenue à 300 Hz.

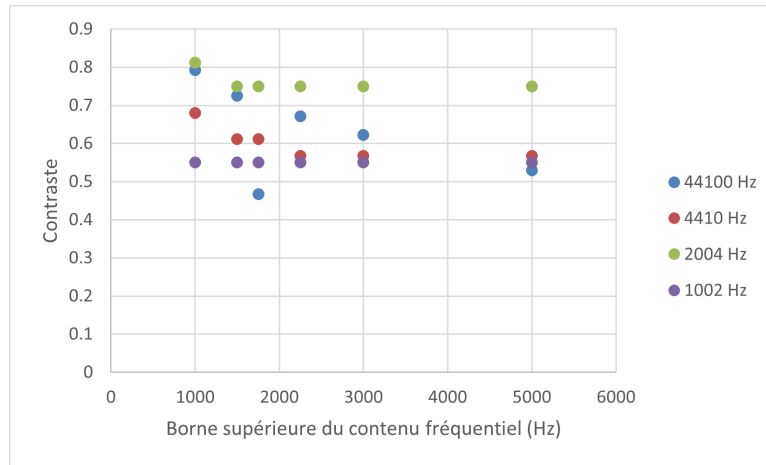


Figure 6 : Contraste en fonction de la fréquence d'échantillonnage (1002, 2004, 4410 et 44100 Hz) et de la fréquence de la borne supérieure du contenu fréquentiel. Ici, la borne inférieure est maintenue à 300 Hz.

En observant les tendances, on voit clairement que pour les fréquences d'échantillonnage inférieures au double de la fréquence maximale ($f < f_{nyquist}$), les variations de résolution et de contraste disparaissent. Par exemple, les résultats obtenus pour une fréquence d'échantillonnage de 1002 Hz sont constants pour toutes les bornes (résolution = 7.17 ± 0.09 cm, contraste = 0.551 ± 0.009). En augmentant la fréquence d'échantillonnage, le même phénomène est observé ; pour une fréquence d'échantillonnage de 2004 Hz, tous les résultats au dessus de $f/2 = 1002$ Hz deviennent constants et pour une fréquence d'échantillonnage de 4410 Hz, tous les résultats au dessus de $f/2 = 2205$ Hz deviennent constants. En dessous de la fréquence d'échantillonnage de Nyquist, les "vraies" fréquences ne sont pas détectées, mais une version "repliée" (*aliasing*) d'elles-mêmes, à plus basses fréquences, peuvent l'être. Le phénomène est illustré dans la figure 7.

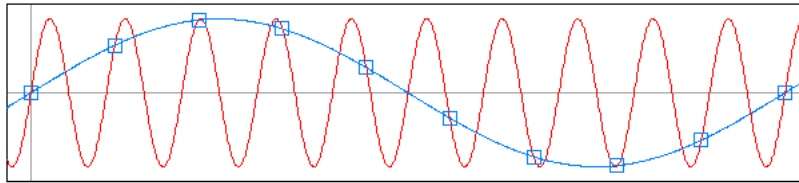


Figure 7 : Exemple du phénomène d'*aliasing* [1]

Par conséquent, il est logique que les fréquences plus grandes que $f_{sample}/2$ ne soient pas détectées, et que de les éliminer suite au filtre n'affecte d'aucune manière le résultat final. Ces tests supportent donc le fait que la fréquence d'acquisition de données ne doit donc jamais être inférieure à la fréquence de Nyquist. Si la composante fréquentielle maximale est connue, la fréquence d'acquisition peut donc être posée au double de celle-ci.

2.5 Incertitudes

L'incertitude associée à l'amplitude du signal, ou l'axe y, est celle qui provient de la résolution de l'ADC. La résolution de l'ADC, δ , dépend du nombre de bits des échantillons, et peut être décrite par :

$$\delta = \frac{\Delta_{max}}{2^n},$$

où Δ_{max} est l'éventail possible des mesures, et n est le nombre de bits. L'incertitude correspond à la moitié de cette valeur, soit $\sigma_{res} = \delta/2$. L'incertitude sur la position de la source du signal est approximée par la moitié de la largeur d'un doigt, $\sigma_{pos} \approx 4 \text{ mm}$. La corrélation des signaux a été déterminée en calculant le produit scalaire. On sait que pour une multiplication, la propagation de l'incertitude se calcule de la manière suivante :

$$\sigma_z = z \sqrt{\frac{\sigma_x^2}{x} + \frac{\sigma_y^2}{y}}.$$

L'incertitude totale sur le produit scalaire est donc :

$$\sigma_{tot} = \sqrt{\sum_i^N \sigma_{z_i}^2},$$

où N est le nombre de points évalués pour les échantillons. Pour déterminer l'incertitude sur les paramètres du fit gaussien, tout en considérant l'impact des incertitudes en x et y , la fonction *scipy.ODR* a été utilisée. Finalement, l'incertitude sur le contraste correspond directement à l'incertitude sur le paramètre de l'amplitude (σ_A), et l'incertitude sur la résolution (FWHM) correspond à :

$$\sigma_{FWHM} = \sqrt{2 \ln 2} \sigma_\sigma,$$

où σ_σ est l'incertitude sur l'écart-type, σ .

3 Codes

Les codes présentés ci-dessous ont été utilisés pour faire fonctionner le piano et pour faire les multiples tests. Ces derniers peuvent nécessiter d'être adaptés pour pouvoir faire des tests spécifiques, ou encore pour décider quelles notes peuvent être jouées par le piano.

3.1 Programme pour modifier le contenu fréquentiel et la fréquence d'échantillonnage

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import json
4 import os
5 from scipy.optimize import curve_fit
6
7 # Charger le fichier JSON (dictionnaire de notes)
8 with open('C:/Users/maxim/OneDrive/Documents/GitHub/piano_project/Wav-Notes/
    notes_dict_1ligne.json', 'r') as f:
9     data = json.load(f)
10 input_filepath = 'Wav-Notes/notes_dict_1ligne.json'
11 # Obtenir le repertoire du fichier original
12 output_directory = os.path.dirname(input_filepath)
13
14 # Parametres a ajuster
15 # liste filtre_bas : [100,100,150,150,200,200,250,250,300,300,350,350,400,400,500,500]
16 # liste filtre_haut :
    [2000,1500,2000,1500,2000,1500,2000,1500,2000,1500,2000,1500,2000,1500,2000,1500]
17 filtre_bas=[550]
18 filtre_haut=[1500]
19 redu=30 #facteur de reduction de la frequence d'echantillonnage
20
21 # Parametres importants

```

```
22 fs = int(44100//redu) # sample rate
23 dt = 0.1 # Intervalle de temps (en secondes)
24 nb_recordings = 1 # nb d'enregistrements par note
25 nb_points = int(dt * fs) # Equivalent en nombre de points pour les indices
26 point_du_tap = 11 # Sert a l'affichage
27
28 notes= ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17']
29 notes_matrix = np.zeros((len(notes) * nb_recordings, nb_points))
30
31 # Transferer les donnees du dictionnaire dans une matrice avec 17 lignes et nb_points
   par ligne
32 i = 0
33 for note, recordings in data.items():
34     for prise in recordings:
35         array = np.array(prise)
36         array_fin=array[:,redu]
37         if len(array_fin)!=nb_points:
38             array_fin=array_fin[:nb_points]
39         notes_matrix[i] = array_fin
40         i += 1
41
42 def dico_filtre_passband(matrice, frequence_minimale, frequence_maximale):
43     # 1. Effectuer la transformee de Fourier rapide (FFT)
44     signal_fft = np.fft.rfft(matrice)
45     frequencies = np.fft.rfftfreq(len(matrice[0]), 1/fs)
46
47     # 2. Creer un filtre passe-bande
48     low_cutoff = frequence_minimale # Frequence de coupure basse (Hz)
49     high_cutoff = frequence_maximale # Frequence de coupure haute (Hz)
50     filter_mask = (frequencies > low_cutoff) & (frequencies < high_cutoff)
51
52     # 3. Appliquer le filtre
53     filtered_fft = signal_fft * filter_mask
54
55     # 4. Revenir au domaine temporel avec la transformee inverse de Fourier
56     filtered_signal = np.fft.irfft(filtered_fft)
57
58     # 5. Remplacer la deuxieme moitie de chaque vecteur par des zeros
59     filtered_signal_cut = np.zeros_like(filtered_signal) # Creer une matrice du meme
   type, remplie de zeros
60     half_point = filtered_signal.shape[1] // 2 # Obtenir le point de coupure (moitie)
61     filtered_signal_cut[:, :half_point] = filtered_signal[:, :half_point]
62
63     # Retourne le nouveau dico avec les signaux filtres
64     return filtered_signal_cut, frequencies, filtered_fft[point_du_tap], signal_fft[
   point_du_tap]
65
66 # Fonction pour creer un nouveau fichier JSON pour chaque niveau de bits
67 def create_modified_json(fbas,fhaut,freq_sampling, output_directory):
68     notes_dict = {}
69
70     for bas,haut in zip(fbas,fhaut):
71         matrice_traitee, frequencies, filtered_fft, signal_fft=dico_filtre_passband(
   notes_matrix,bas,haut)
72
73     # On parcourt les notes et les enregistrements pour remplir le dictionnaire
74     for i, note in enumerate(notes):
75         recordings = []
76         for j in range(nb_recordings):
```

```
77         # On suppose que chaque enregistrement correspond a une ligne dans
notes_matrix
78         recordings.append(matrice_traitee[i * nb_recordings + j].tolist()) #
Convertir en liste
79         notes_dict[note] = recordings
80
81     for bas, haut in zip(fbas, fhaut):
82         # Generer le nom de fichier pour chaque duo de filtre
83         output_filename = os.path.join(output_directory, f'fs={freq_sampling}-fbas={bas
}-fhaut={haut}.json')
84
85         # Sauvegarder le nouveau dictionnaire dans un fichier JSON
86         with open(output_filename, 'w') as outfile:
87             json.dump(notes_dict, outfile, indent=4)
88
89 # Creer un fichier JSON pour chaque niveau de bits dans le meme repertoire que le
fichier original
90 create_modified_json(filtre_bas, filtre_haut, fs, output_directory)
91
92 #La suite du code n'est pas utile en soi mais permet de confirmer que ca donne la bonne
chose
93 sign, frequencies, filtered_fft, signal_fft=dico_filtre_passband(notes_matrix, filtre_bas
[0], filtre_haut[0])
94 signal_post_filtre=sign[point_du_tap]
95 signal=notes_matrix[point_du_tap]
96
97 # Generer le temps
98 t1 = np.linspace(0, 1, len(signal), endpoint=False) # Intervalle de temps
99 t2 = np.linspace(0, 1, len(signal_post_filtre), endpoint=False) # Intervalle de temps
100 #f1, f2 = 50, 200 # Frequences des sinusoides
101 #signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)
102
103 # Afficher les resultats
104 plt.figure(figsize=(10, 6))
105
106 # Affichage du signal original
107 plt.subplot(2, 2, 1)
108 plt.plot(t1, signal)
109 plt.title("Signal original")
110 plt.xlabel("Temps [s]")
111 plt.ylabel("Amplitude")
112
113 # Spectre de frequence original
114 plt.subplot(2, 2, 2)
115 plt.plot(frequencies[:fs//2], np.abs(signal_fft)[:fs//2])
116 plt.title("Spectre de frequence original")
117 plt.xlabel("Frequence [Hz]")
118 plt.ylabel("Amplitude")
119
120 # Spectre de frequence filtre
121 plt.subplot(2, 2, 4)
122 plt.plot(frequencies[:fs//2], np.abs(filtered_fft)[:fs//2])
123 plt.title("Spectre de frequence filtre")
124 plt.xlabel("Frequence [Hz]")
125 plt.ylabel("Amplitude")
126
127 # Signal filtre
128 plt.subplot(2, 2, 3)
129 plt.plot(t2, signal_post_filtre.real)
```

```
130 plt.title("Signal filtre")
131 plt.xlabel("Temps [s]")
132 plt.ylabel("Amplitude")
133
134 plt.tight_layout()
135 plt.show()
```

3.2 Programme pour changer le nombre de bits des échantillons

```
1 import json
2 import os
3 import numpy as np
4
5 # Charger le fichier JSON (dictionnaire)
6 input_filepath = 'Wav-Notes/notes_dict_1ligne.json'
7 with open(input_filepath, 'r') as f:
8     data = json.load(f)
9
10 # Obtenir le repertoire du fichier original
11 output_directory = os.path.dirname(input_filepath)
12
13 # Fonction pour reduire la precision d'un signal en fonction du nombre de bits
14 def reduce_precision(signal, n_bits):
15     if n_bits == 1:
16         # Cas special pour 1 bit : toutes les valeurs negatives deviennent 0
17         return [1 if value > 0 else 0 for value in signal]
18     elif n_bits > 1:
19         levels = 2**(n_bits - 1) # Utiliser n_bits - 1 pour tenir compte du bit de
20         # signe
21         # Quantifier le signal dans le nombre de niveaux approprié
22         signal_quantified = np.round(np.array(signal) * (levels // 2)) / (levels // 2)
23     else:
24         signal_quantified = np.zeros_like(signal) # Tout est ramene a zero pour 0 bit
25
26     return signal_quantified.tolist() # Convertir en liste pour garder le format JSON
27
28 # Fonction pour creer un nouveau fichier JSON pour chaque niveau de bits
29 def create_modified_json(data, n_bits, output_directory):
30     # Creer un nouveau dictionnaire avec les signaux modifies
31     modified_data = {}
32
33     for note, vecteurs in data.items():
34         modified_data[note] = [reduce_precision(vecteur, n_bits) for vecteur in vecteurs]
35
36     # Generer le nom de fichier pour chaque niveau de bits
37     output_filename = os.path.join(output_directory, f'modified_signal_{n_bits}bit.json')
38
39     # Sauvegarder le nouveau dictionnaire dans un fichier JSON
40     with open(output_filename, 'w') as outfile:
41         json.dump(modified_data, outfile, indent=4)
42
43 # Creer un fichier JSON pour chaque niveau de bits dans le meme repertoire que le
44 # fichier original
45 create_modified_json(data, 16, output_directory)
46 create_modified_json(data, 8, output_directory)
47 create_modified_json(data, 6, output_directory)
```

```

46 create_modified_json(data, 4, output_directory)
47 create_modified_json(data, 3, output_directory)
48 create_modified_json(data, 2, output_directory)
49 create_modified_json(data, 1, output_directory)
50 create_modified_json(data, 0, output_directory)

```

3.3 Programme pour calculer la résolution et le contraste pour un dictionnaire de notes

```

1 import numpy as np
2 from scipy.odr import ODR, Model, RealData
3 import json
4 import pandas as pd
5 from scipy.optimize import curve_fit
6 import matplotlib.pyplot as plt
7 print('Librairies importees')
8
9 # liste des noms de dictionnaires JSON a charger
10 fichiers=['notes_dict_1ligne', 'modified_signal_1bit', 'modified_signal_2bit', '
    modified_signal_3bit'
11           , 'modified_signal_4bit', 'modified_signal_6bit', 'modified_signal_8bit', '
    modified_signal_16bit']
12
13 # Charger les fichiers JSON
14 def lecteur():
15     encyclopedie=[]
16     for nom in fichiers:
17         with open(f'Wav-Notes/{nom}.json', 'r') as f:
18             encyclopedie.append(json.load(f))
19     return encyclopedie
20
21 # Fonction gaussienne avec floor ajustable pour curve_fit
22 def gaussian_with_floor(x, A, mu, sigma, floor):
23     return A * np.exp(-((x - mu) ** 2) / (2 * sigma ** 2)) + floor
24
25 # Fonction pour ajuster avec des bornes (curve_fit)
26 def fit_gaussian_with_bounds(corr_data, xaxis, yerr):
27     x_data = xaxis # np.arange(len(corr_data))
28     initial_guess = [np.max(corr_data), np.argmax(corr_data), np.std(corr_data), np.mean(
    corr_data)] # [amplitude, mean, sigma, offset]
29
30     # Contraintes sur les bornes pour que sigma > 0 et floor proche de la moyenne des
    donnees
31     bounds = ([0, 0, 0, np.mean(corr_data) - 0.09], [np.inf, len(corr_data), np.inf, np.
    mean(corr_data) + 0.09])
32
33     # Utilisation de curve_fit avec la nouvelle fonction gaussienne
34     params, pcov = curve_fit(gaussian_with_floor, x_data, corr_data, p0=initial_guess,
    sigma=yerr, absolute_sigma=True, bounds=bounds, maxfev
35     =10000)
36
37     perr = np.sqrt(np.diag(pcov)) # Erreurs sur les parametres ajustes
38     return params, perr, x_data
39
40 # Fonction pour calculer la correlation du curve_fit
41 def correlativeur_et_curve_fit_gaussien(data, point_du_tap):
42     #Remarque sur le code a Marielou
43     # Parametres importants
44     nb_points = len(data['1'][0])

```

```

45 dt = 0.1
46 fs=int(nb_points/dt) # sample rate
47 nb_recordings = 1 # nb d'enregistrements par note
48
49 notes = ['1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16','17']
50 matrice = np.zeros((len(notes) * nb_recordings, nb_points))
51
52 i = 0
53 for note, recordings in data.items():
54     for prise in recordings:
55         array = np.array(prise)
56         array_normalise = array / np.max(array)
57         matrice[i] = array_normalise
58         i += 1
59
60 signaux_reference = matrice
61 signal = matrice[point_du_tap]
62 position = 3*10**(-2) + 1.5*np.arange(0, 17)*10**(-2)
63 correlation = np.dot(signaux_reference, signal)/np.max(np.dot(signaux_reference,
64 signal))
65
66 nb_bits = 32
67 range = 2
68 err_ampl = (range/(2**(nb_bits)))/2
69
70 # Propagation de l'erreur sur le produit scalaire de la correlation
71 yerr = []
72 for element in signaux_reference:
73     # Remplacer les zeros par une petite valeur pour eviter la division par zero
74     signal_safe = np.where(signal == 0, 1e-10, signal)
75     element_safe = np.where(element == 0, 1e-10, element)
76
77     # Calculer l'erreur de multiplication
78     err_multiplication = (signal_safe * element_safe) * np.sqrt((err_ampl /
79 signal_safe) ** 2 + (err_ampl / element_safe) ** 2)
80
81     # Verifier si err_multiplication contient des NaN ou des infinis
82     if np.any(np.isnan(err_multiplication)) or np.any(np.isinf(err_multiplication)):
83         print("Attention : err_multiplication contient des NaN ou des infinis.")
84         continue # Passer a l'iteration suivante
85
86     # Calculer l'erreur totale
87     err_prod = np.sqrt(np.sum(err_multiplication ** 2))
88     yerr.append(err_prod)
89 #yerr=1*correlation
90
91 params, perr, x_data = fit_gaussian_with_bounds(correlation, position, yerr)
92
93 amplitude_fit, mean_fit, sigma_fit, offset_fit = params
94 amplitude_err, mean_err, sigma_err, offset_err = perr
95
96 # La resolution est convertie en cm (chaque point est espace de 1,5 cm)
97 resolution = 1.5 * np.log(2) * np.sqrt(2) * sigma_fit
98 resolution_err = 1.5 * np.log(2) * np.sqrt(2) * sigma_err
99
100 max_diff = amplitude_fit-offset_fit
101 max_diff_err = amplitude_err-offset_err

```

```
101     return {
102         "resolution": resolution,
103         "resolution_err": resolution_err,
104         "max_diff": max_diff,
105         "max_diff_err": max_diff_err,
106     }
107
108 # Fonction gaussienne ODR
109 def gaussian_with_floor_constrained(p, x):
110     A, mu, log_sigma, floor = p
111     sigma = np.exp(log_sigma) # sigma > 0 en utilisant la transformation exponentielle
112     return A * np.exp(-((x - mu) ** 2) / (2 * sigma ** 2)) + floor
113
114 # Fonction pour calculer la correlation ODR
115 def correlateur_ODR(data, point_du_tap, nb_bit):
116     # Parametres importants
117     nb_points = len(data['1'][0])
118     dt = 0.1
119     fs=int(nb_points/dt) # sample rate
120     nb_recordings = 1 # nb d'enregistrements par note
121
122     notes = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17']
123     matrice = np.zeros((len(notes) * nb_recordings, nb_points))
124
125     i = 0
126     for note, recordings in data.items():
127         for prise in recordings:
128             array = np.array(prise)
129             array_normalise = array / np.max(array)
130             matrice[i] = array_normalise
131             i += 1
132
133     signaux_reference = matrice
134     signal = matrice[point_du_tap]
135     position = 3*10**(-2) + 1.5*np.arange(0, 17)*10**(-2)
136     correlation = np.dot(signaux_reference, signal)/np.max(np.dot(signaux_reference,
137                             signal))
138
139     nb_bits = nb_bit
140     range = 2
141     err_ampl = (range/(2**(nb_bits)))/2
142     err_position = 4e-3
143
144     # Propagation de l'erreur sur le produit scalaire de la correlation
145     err_prod_ampl = []
146     for element in signaux_reference:
147         # Remplacer les zeros par une petite valeur pour eviter la division par zero
148         signal_safe = np.where(signal == 0, 1e-10, signal)
149         element_safe = np.where(element == 0, 1e-10, element)
150
151         # Calculer l'erreur de multiplication
152         err_multiplication = (signal_safe * element_safe) * np.sqrt((err_ampl /
153             signal_safe) ** 2 + (err_ampl / element_safe) ** 2)
154
155         # Verifier si err_multiplication contient des NaN ou des infinis
156         if np.any(np.isnan(err_multiplication)) or np.any(np.isinf(err_multiplication)):
157             print("Attention : err_multiplication contient des NaN ou des infinis.")
158             continue # Passer a l'iteration suivante
```

```
157
158     # Calculer l'erreur totale
159     err_prod = np.sqrt(np.sum(err_multiplication ** 2))
160     err_prod_ampl.append(err_prod)
161
162     # Utilisation de ODR pour faire le fit gaussien avec sigma toujours positif
163     data = RealData(position, correlation, sx=err_position, sy=err_prod_ampl)
164     model = Model(gaussian_with_floor_constrained)
165
166     guess_initial = [np.max(correlation), np.mean(position), np.log(np.std(position)),
167                      np.mean(correlation)]
168     odr = ODR(data, model, beta0=guess_initial)
169     output = odr.run()
170
171     # Parametres optimaux et matrice de covariance
172     A_opt, mu_opt, log_sigma_opt, floor_opt = output.beta
173     sigma_opt = np.exp(log_sigma_opt) # Revenir a sigma
174
175     resolution = np.sqrt(2 * np.log(2)) * sigma_opt
176     resolution_err = np.sqrt(2 * np.log(2)) * sigma_opt * np.sqrt(output.cov_beta[2, 2])
177
178     contraste = A_opt - np.mean(correlation)
179     contraste_err = 2 * np.sqrt(output.cov_beta[0, 0])
180
181     return {
182         "resolution": resolution,
183         "resolution_err": resolution_err,
184         "max_diff": contraste,
185         "max_diff_err": contraste_err,
186     }
187
188     # Ajuster les donnees avec la fonction gaussienne avec plancher
189     params, perr, x_data = fit_gaussian_with_offset_and_errors(corr_data, yerr)
190
191     # Extraction des parametres ajustes
192     amplitude_fit, mean_fit, sigma_fit, offset_fit = params
193     amplitude_err, mean_err, sigma_err, offset_err = perr
194
195     # La resolution est convertie en cm (chaque point est espace de 1,5cm)
196     resolution = 1.5 * np.log(2) * np.sqrt(2) * sigma_fit
197     resolution_err = 1.5 * np.log(2) * np.sqrt(2) * sigma_err
198
199     # Calcul de la difference entre le maximum de la gaussienne et l'offset
200     max_diff = amplitude_fit
201     max_diff_err = amplitude_err # Incertitude sur la difference est celle de l'
202     amplitude
203
204     # Retourner les resultats
205     return {
206         "resolution": resolution,
207         "resolution_err": resolution_err,
208         "max_diff": max_diff,
209         "max_diff_err": max_diff_err,
210     }
211
212     # Creer une liste pour stocker les resultats
213     results_list = []
214     dictionaries_to_process = lecteur() # Ajoute les autres dictionnaires ici
215     bit_names = ['Original', '1bit', '2bit', '3bit', '4bit', '6bit', '8bit', '16bit']
```



```

214 bit_qty = [32, 1, 2, 3, 4, 6, 8, 16] # bit_qty[idx]
215
216 for idx, current_data in enumerate(dictionaries_to_process):
217     a1 = []
218     a2 = []
219     a3 = []
220     a4 = []
221     for i in [7, 8, 9, 10, 11]:
222         corr_data = correlateur_ODR(current_data, i, bit_qty[idx])
223
224         # Analyser les ajustements
225         results = corr_data
226         a1.append(results["resolution"])
227         a2.append(results["resolution_err"])
228         a3.append(results["max_diff"])
229         a4.append(results["max_diff_err"])
230
231     resolution = np.mean(a1)
232     resolution_err = np.mean(a2)
233     contraste = np.mean(a3)
234     contraste_err = np.mean(a4)
235
236     # Ajouter les resultats a la liste, en incluant les noms des bits
237     results_list.append({
238         "Dictionnaire": bit_names[idx],
239         "Resolution": resolution,
240         "Erreur Resolution": resolution_err,
241         "Contraste": contraste,
242         "Erreur Contraste": contraste_err,
243     })
244
245 # Convertir les resultats en DataFrame
246 results_df = pd.DataFrame(results_list)
247
248 # Exporter les resultats en fichier Excel
249 results_df.to_excel('resultats_nouveaux_points_nb_bit.xlsx', index=False)
250
251 print("Analyse terminee et resultats exportes vers 'resultats'.")

```

3.4 Programme pour enregistrer un dictionnaire de notes

```

1 import sounddevice as sd
2 import numpy as np
3 import json
4
5 # temps pour taper la note
6 seconds = 2
7
8 # freq d'acquisition par default
9 fs = 44100
10
11 default = True # Si cette option est utilisee, le micro/speaker par default est utilise
12 devices = sd.query_devices()
13
14 if not default:
15     InputStr = "Choisir le # correspondant au micro parmi la liste: \n"
16     OutputStr = "Choisir le # correspondant au speaker parmi la liste: \n"
17     for i in range(len(devices)):

```

```

18         if devices[i]['max_input_channels']:
19             InputStr += ('%d : %s \n' % (i, ''.join(devices[i]['name'])))
20         if devices[i]['max_output_channels']:
21             OutputStr += ('%d : %s \n' % (i, ''.join(devices[i]['name'])))
22     DeviceIn = input(InputStr)
23     DeviceOut = input(OutputStr)
24
25     sd.default.device = [int(DeviceIn), int(DeviceOut)]
26
27     # liste de notes a enregistrer
28     notes = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17']
29
30     notes_dict = {note: None for note in notes}
31     dt = 1e-1 # Intervalle de temps (en secondes)
32     nb_points = int(dt*fs) # Equivalent en nombre de points pour les indices
33
34     # enregistrement de chaque note
35     for note in notes:
36         recordings = []
37         print(f'Enregistrement de la note', note)
38
39         # ecoute de la note
40         myrecording = sd.rec(int(seconds * fs), samplerate=fs, channels=1)
41         sd.wait()
42         print(f'Enregistrement fini.')
43
44         # Trouver l'amplitude maximale en valeur absolue
45         max_amplitude = np.max(abs(myrecording))
46         threshold = max_amplitude / 10 # Definir le seuil
47         print(threshold)
48
49         # Creer la fenetre utilisee pour le signal
50         for index, value in enumerate(myrecording):
51             if value >= threshold:
52                 start_signal = index
53                 break
54
55         cut_signal = myrecording[start_signal:(start_signal + nb_points)].flatten()
56
57         # Normalisation du signal
58         norm_cut_signal = cut_signal / max_amplitude
59
60         # Rajouter le nouveau array a la liste 'recordings'
61         recordings.append(norm_cut_signal.tolist())
62
63         # Transformer la liste en array
64         nom_note = note
65         notes_dict[nom_note] = recordings
66
67     # sauvegarde du dictionnaire en JSON
68     with open('notes_dict_1ligne.json', 'w') as json_file:
69         json.dump(notes_dict, json_file)

```

3.5 Programme pour jouer du piano en temps réel

```

1 import sounddevice as sd
2 import numpy as np
3 import threading

```

```
4 import matplotlib.pyplot as plt
5 import time
6 from collections import deque
7 import pygame
8 from pydub import AudioSegment
9 import os
10 import json
11 import math
12
13 # chargement du dictionnaire
14 with open('Wav-Notes\\notes_dict_jo.json', 'r') as file:
15     data = json.load(file)
16
17 # parametres audio
18 fs = 44100
19 dt = 0.1 # Intervalle de temps (en secondes)
20 nb_recordings = 1 # Combien de signaux par note
21 nb_points = int(dt * fs)
22
23 # liste de notes
24 notes = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15',
25         '16', '17']
26 notes_matrix = np.zeros((len(notes) * nb_recordings, nb_points))
27
28 # Conversion des donnees en arrays et reformatage de la memoire
29 i = 0
30 for note, recordings in data.items():
31     for element in recordings:
32         array = np.array(element)
33         notes_matrix[i] = array
34         i += 1
35
36 notes_dict = {
37     '1': 'c3.wav',
38     '2': 'c-3.wav',
39     '3': 'd3.wav',
40     '4': 'd-3.wav',
41     '5': 'e3.wav',
42     '6': 'f3.wav',
43     '7': 'f-3.wav',
44     '8': 'g3.wav',
45     '9': 'g-3.wav',
46     '10': 'a4.wav',
47     '11': 'a-4.wav',
48     '12': 'b4.wav',
49     '13': 'c4.wav',
50     '14': 'c-4.wav',
51     '15': 'd4.wav',
52     '16': 'd-4.wav',
53     '17': 'e4.wav'
54 }
55
56 pygame.mixer.init()
57
58 # THREAD : fonction pour jouer l'audio d'une note
59 def jouer_note(note):
60     if note in notes_dict:
61         fichier_note = f'Wav-Notes\\{notes_dict[note]}'
```

```
62         if os.path.isfile(fichier_note):
63             # Load the WAV file in memory
64             note_sound = AudioSegment.from_wav(fichier_note)
65
66             # Play the note using pygame
67             son = pygame.mixer.Sound(fichier_note)
68             son.play()
69             pygame.time.wait(int(note_sound.duration_seconds * 1000)) # attendre fin de
l'audio
70         else:
71             print(f"Le fichier {fichier_note} n'existe pas.")
72     else:
73         print("Note non reconnue.")
74
75 # test de note
76 jouer_note('c3')
77
78 # parametres de detection de signal audio
79 threshold = 0.01 # amplitude minimale pour signaler une impulsion
80 spike_detected = False # flag pour savoir si un signal est detecte
81 capture_duration = 0.15 # temps de capture d'audio post impulsion
82 buffer_size = fs # taille du buffer qui contient le signal audio roulant
83 signal_buffer = np.zeros(buffer_size)
84
85 # Au besoin, fonction pour visualiser le signal audio d'une impulsion
86 def plot_data(data):
87     t = np.linspace(0, dt, len(data))
88     plt.plot(t, data)
89     plt.xlabel('Time [s]')
90     plt.ylabel('Amplitude')
91     plt.title('Signal After Spike')
92     plt.show()
93
94 # Lecture d'audio en temps reel
95 def audio_callback(indata, frames, time, status):
96     global spike_detected, signal_buffer
97
98     if status:
99         print(status)
100
101     # aplatissage du data entrant
102     audio_data = indata[:, 0]
103
104     # roulement du signal dans le buffer
105     signal_buffer = np.roll(signal_buffer, -frames)
106     signal_buffer[-frames:] = audio_data
107
108     # Analyse du buffer pour l'impulsion
109     if not spike_detected and np.max(audio_data) > threshold:
110         spike_detected = True
111         print("Spike detected!")
112
113     # Si detection, lance le thread d'analyse de signal
114     capture_thread = threading.Thread(target=signal_analysis)
115     capture_thread.start()
116
117 # THREAD : Analyse de signal
118 def signal_analysis():
119     global spike_detected
```

```
120
121     time.sleep(capture_duration) # laisse le buffer prendre la suite du signal post
impulsion
122
123     # Capture du signal de l'impulsion dans le buffer
124     post_spike_data = np.copy(signal_buffer)[int(fs - fs * capture_duration - 800):]
125
126     #####
127     # Traitement de donnees
128     #####
129
130     data_max_amp = np.max(abs(post_spike_data))
131     data_threshold = data_max_amp / 10
132
133     # Creer la fenetre utilisee pour le signal
134     for index, value in enumerate(post_spike_data):
135         if value >= data_threshold:
136             start_signal = index
137             break
138     cut_data = post_spike_data[start_signal:(start_signal + int(dt * fs)).flatten()
139
140     # Plot the data
141     # plot_data(cut_data)
142
143     # Normalisation du signal
144     norm_cut_data = cut_data / data_max_amp
145
146     # Transformer la liste en array
147     signal_array = np.array(norm_cut_data)
148
149     # Produit scalaire (correlation) entre les donnees de training et le signal test
150     scalar_prod = np.dot(notes_matrix, signal_array)
151
152     # Trouver l'indice de la valeur max du produit scalaire et trouver sa note
correspondante
153     index_max = np.argmax(scalar_prod)
154     note_index = index_max // nb_recordings
155     print(scalar_prod)
156
157     note = notes[note_index]
158     if note:
159         print(f"Playing note: {note}")
160         # Lance le thread pour jouer la note identifiee
161         play_note_thread = threading.Thread(target=jouer_note, args=(note,))
162         play_note_thread.start()
163
164     # Reset du flag pour continuer a lire des impulsions
165     spike_detected = False
166
167 # Lancement de la lecture d'audio en continu
168 with sd.InputStream(callback=audio_callback, samplerate=fs, channels=1):
169     print("Recording... (Press Ctrl+C to stop)")
170     while True:
171         time.sleep(0.001) # Keep the main loop running
```

Références

- [1] NATIONAL INSTRUMENTS. *Acquiring an Analog Signal : Bandwidth, Nyquist Sampling Theorem, and Aliasing*. 2024. URL : `%5Curl%7Bhttps://www.ni.com/en/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/acquiring-an-analog-signal--bandwidth--nyquist-sampling-theorem-.html#:~:text=The%20Nyquist%20Sampling%20Theorem%20explains,the%20Nyquist%20frequency%2C%20fN.%7D`.